writeup.md

Vehicle Detection Project

writeup

The goals / steps of this project are the following:

- Perform a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images and train a classifier Linear SVM classifier
- Optionally, you can also apply a color transform and append binned color features, as well as histograms of color, to your HOG feature vector.
- Note: for those first two steps don't forget to normalize your features and randomize a selection for training and testing.
- Implement a sliding-window technique and use your trained classifier to search for vehicles in images.
- Run your pipeline on a video stream (start with the test_video.mp4 and later implement on full project_video.mp4) and create a heat map of recurring detections frame by frame to reject outliers and follow detected vehicles.
- Estimate a bounding box for vehicles detected.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

General code organization

The code is divided in three files:

- `functions.py` : functions to use in processing the images
- `training.py` : code to train the classifier
- `detection.py` : code that uses the trained classifier to process images/videos

The same feature extraction code is used by both files to make sure the detection uses the same features which were used to train the classifier.

Images and features

1. Explain how (and identify where in your code) you extracted HOG features from the training images.

To compare images, I extracted several features from each:

- color binning
- color histograms
- histogram of oriented gradients (HOG)

code to extract these features from a single image is in function `img_features()` (in file `functions.py`):

```
def img_features(img, spatial_size, color_space, hist_bins, orient, pix_per_cell, cell_per_block, hog_channel, windo
    # resize the image to the processing size. It must be the same as in training to get same feature vector length
    feature_image = cv2.resize(img, window_size)
```

```
    file_features = []
    if spatial_feat:
        spatial_features = bin_spatial(feature_image, color_space=color_space, size=spatial_size)
        # print("spatial features: ",len(spatial_features))
        file_features.append(spatial_features)
    if hist_feat:
        # Apply color_hist()
        hist_features = color_hist(feature_image, nbins=hist_bins)
        # print("hist features: ",len(hist_features))
        file_features.append(hist_features)
    if hog_feat:
        if hog_channel == 'ALL':
            hog_features = []
            for channel in range(feature_image.shape[2]):
                hog_features.extend(get_hog_features(feature_image[:, :, channel],
                                                     orient, pix_per_cell, cell_per_block,
                                                     vis=False, feature_vec=True))
        else:
            hog_features = get_hog_features(feature_image[:, :, hog_channel], orient,
                                            pix_per_cell, cell_per_block, vis=False, feature_vec=True)

        # print("hog features: ",len(hog_features))
        file_features.append(hog_features)
    return np.concatenate((file_features))
```

this function gets the defining parameters for each feature, and returns a single vector with features concatenated. The parameters are used in other parts of the code so I will explain them here:

`img` : image to process

`spatial_size` : (dx, dy) size of the window to get color binning

`color_space` : String, one of RGB, HSV, LUV, HLS, YUV, YCrCb. Color space in which to get color binning. The image will be converted to this color space and resized to `spatial_size` to get the feature.

`hist_bins` : number of bins to consider for color histograms.

`orient` : number of orientations to consider for HOG feature.

`pix_per_cell` : number of pixels per cell for HOG extraction

`cell_per_block` : number of cells per block to consider for HOG extraction

`hog_channel` : channel to extract HOG feature from. May be 0, 1, 2, or 'ALL'

`window_size` : (dx, dy) size of the window used to train the classifier.

`spatial_feat` : boolean. If true, spatial binning feature is extracted

`hist_feat` : boolean. If true, histogram features are extracted

`hog_feat` : boolean. If true, HOG features are extracted

Last three parameters govern which features are extracted from the image. Let's take a look at each one:

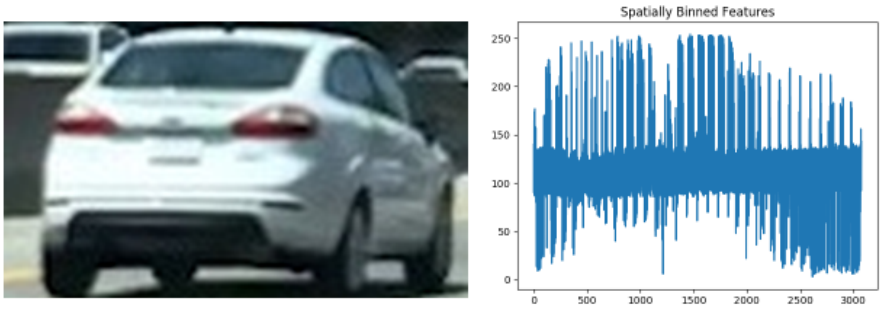
Spatial binning

The spatial binning consists on taking the image, resize it to a suitable size, and 'stretch' it to a single vector. The code is simple enough:

```
def bin_spatial(img, color_space='RGB', size=(64, 64)):
    image = convertColor(img, color_space)
    features = cv2.resize(image, size).ravel()
    return features
```

Optionally, the image can be converted to a different color space before flatten it. Function `convertColor()` takes the original RGB image and convert it to the desired color space:

```
def convertColor(image, color_space):
    if color_space == 'RGB':
        feature_image = image
    elif color_space == 'HSV':
        feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    elif color_space == 'LUV':
        feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2LUV)
    elif color_space == 'HLS':
        feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
    elif color_space == 'YUV':
        feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2YUV)
    elif color_space == 'YCrCb':
        feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2YCrCb)
    else:
        feature_image = np.copy(image)
```



The original image is always loaded as RGB. I used a function to load the image and convert if necessary (OpenCV uses BGR, for example):

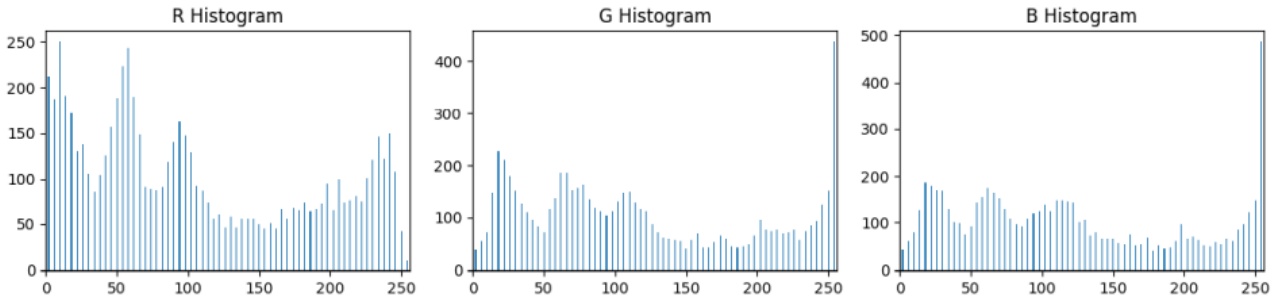
```
def loadRGBImage(path):
    # if using matplotlib.image, the image is already RGB but scaled from 0 to 1
    # img = mpimg.imread(path)
    img = cv2.imread(path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    return img
```

Color histograms

Another feature that can be useful is the color histograms: for each color channel, we compute the histogram -distribution of intensity- and group the values in 'bins':

```
def color_hist(img, nbins=32, bins_range=(0, 256)):
    # Compute the histogram of the color channels separately
    channel1_hist = np.histogram(img[:, :, 0], bins=nbins, range=bins_range)
    channel2_hist = np.histogram(img[:, :, 1], bins=nbins, range=bins_range)
    channel3_hist = np.histogram(img[:, :, 2], bins=nbins, range=bins_range)
    # Concatenate the histograms into a single feature vector
    hist_features = np.concatenate((channel1_hist[0], channel2_hist[0], channel3_hist[0]))
    return hist_features
```

This function returns a single vector with red, green and blue histograms concatenated.

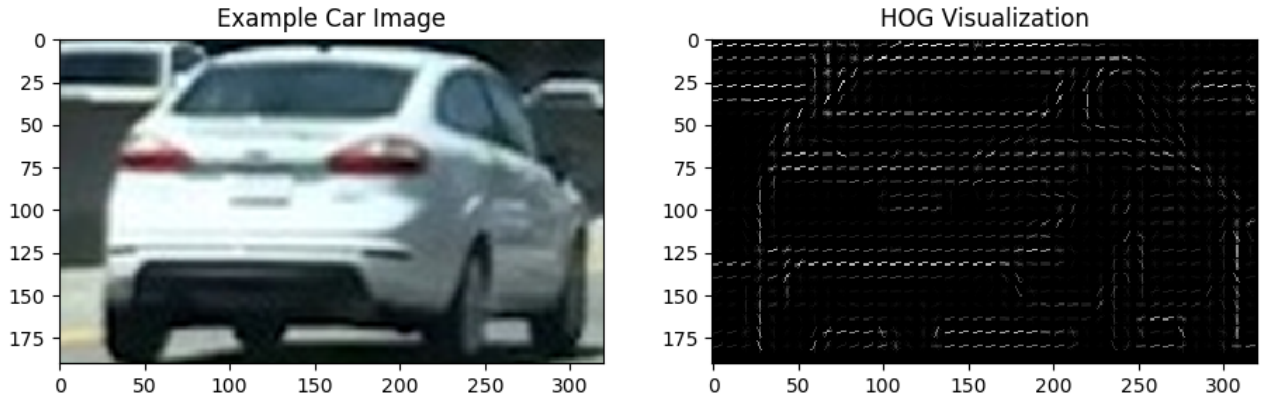


HOG features

The Histogram of Oriented Gradients is another feature that prove to be useful in object detection. The technique counts occurrences of gradient orientation in localized portions of an image. I used the implementation in scikit-image package. According to the documentation (http://scikit-image.org/docs/dev/auto_examples/features_detection/plot_hog.html), the algorithm is the following:

1. (optional) global image normalisation
2. computing the gradient image in x and y
3. computing gradient histograms
4. normalising across blocks
5. flattening into a feature vector

Basically, it computes the gradients of portions of the image (cells), combine them into blocks, and compute the histogram of the gradient orientations on those. Then the block descriptors are combined in a vector, and that vector is what we use as feature for the image.



2. Explain how you settled on your final choice of HOG parameters.

By trial. I tested different values for the parameters `orient`, `pix_per_cell`, `cell_per_block`, and `hog_channel`; the values that worked better for this project are the same used in the lesson:

```
orient = 9
pix_per_cell = 8
cell_per_block = 2
hog_channel = "ALL"
```

Finally, all these features are combined in a single long vector which is used to train the classifier by applying it to training, labeled images of cars and non-cars.

Training

3. Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

I used the training data provided by Udacity in their 'smallset' variants, augmented somehow by images taken manually by me from the test and project videos to aid in the detection. The filenames are loaded in separate arrays for cars and non-cars in the function `getTrainingImageLists`:

```
def getTrainingImageLists(smallset=True, maxImages=0):
    print("Reading image filenames")
    if smallset:
        dir_veh = r'D:\archi\ernesto\cursos\self-driving car\proj5\vehicles_smallset\**\*.jpeg'
        dir_notveh = r'D:\archi\ernesto\cursos\self-driving car\proj5\non-vehicles_smallset\**\*.jpeg'
    else:
        dir_veh = r'D:\archi\ernesto\cursos\self-driving car\proj5\vehicles\**\*.png'
        dir_notveh = r'D:\archi\ernesto\cursos\self-driving car\proj5\non-vehicles\**\*.png'

    images = glob.glob(dir_veh, recursive=True)
    cars = []
    i = 0
    for image in images:
        if maxImages > 0 and i >= maxImages:
            break
        cars.append(image)
        i += 1

    i = 0
    images = glob.glob(dir_notveh, recursive=True)
    notcars = []
    for image in images:
        if maxImages > 0 and i >= maxImages:
            break
        notcars.append(image)
        i += 1

    return cars, notcars
```

Examples of car images used to train

Not cars

This function allows me to select whether to load the smallset of the full dataset, which I also tried. The `maxImages` parameter put a limit to the number of filenames to load, as my machine ran out of memory for large loads. I did try the big datasets, but the results from the small ones was good enough and the time to process all of them to train the classifier was too much, so I decanted for the small set.

The training starts by loading the images and computing their feature vectors:

```
cars, notcars = getTrainingImageLists(smallset=True, maxImages=4000)
car_features = extract_features(cars, color_space=color_space,
                                spatial_size=spatial_size, hist_bins=hist_bins,
                                orient=orient, pix_per_cell=pix_per_cell,
                                cell_per_block=cell_per_block,
                                hog_channel=hog_channel, window_size=image_size,
                                spatial_feat=spatial_feat, hist_feat=hist_feat, hog_feat=hog_feat)
notcar_features = extract_features(notcars, color_space=color_space,
                                   spatial_size=spatial_size, hist_bins=hist_bins,
                                   orient=orient, pix_per_cell=pix_per_cell,
                                   cell_per_block=cell_per_block,
                                   hog_channel=hog_channel, window_size=image_size,
                                   spatial_feat=spatial_feat, hist_feat=hist_feat, hog_feat=hog_feat)
```

the features are collected in lists by the function `extract_features` :

```
def extract_features(imgs, color_space='RGB', spatial_size=(64, 64),
                     hist_bins=32, orient=9,
                     pix_per_cell=8, cell_per_block=2, hog_channel=0, window_size=(64,64),
                     spatial_feat=True, hist_feat=True, hog_feat=True):
    features = []
    for file in imgs:
        # load file always in RGB
        feature_image = loadRGBImage(file)
        # and compute its feature vector
        file_features = img_features(img=feature_image, spatial_size=spatial_size, color_space=color_space,
                                     hist_bins=hist_bins, orient=orient, pix_per_cell=pix_per_cell,
                                     cell_per_block=cell_per_block, hog_channel=hog_channel, window_size=window_size,
                                     spatial_feat=spatial_feat, hist_feat=hist_feat, hog_feat=hog_feat)
        features.append(file_features)
    return features
```

then the features vector is normalized using a StandardScaler from scikit-learn package:

```
X = np.vstack((car_features, notcar_features)).astype(np.float64)
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X
scaled_X = X_scaler.transform(X)
```

along with the features, we need the corresponding labels to train the classifier. The labels are codified as *car* (1) or *not-car* (0), and the labels vector is easily created on the fly:

```
y = np.hstack((np.ones(len(car_features)), np.zeros(len(notcar_features))))
```

Next I divide the dataset in a training portion (80% of the records) and a test portion using again scikit-learn:

```
X_train, X_test, y_train, y_test = train_test_split(scaled_X, y, test_size=0.2, random_state=rand_state)
```

and finally use the data to train and test the classifier, a SVM linear from scikit-learn:

```
svc = LinearSVC()
svc.fit(X_train, y_train)
print('Test Accuracy of SVC = ', round(svc.score(X_test, y_test), 4))
```

I get an accuracy of around 0.97:

```
cars: 1297, not cars: 1125
Using: 9 orientations 8 pixels per cell and 2 cells per block
```

```
Feature vector length: 17628
X_train shape:  (1937, 17628)
y_train shape:  (1937,)
X_test shape:  (485, 17628)
y_test shape:  (485,)
Training...
4.3 Seconds to train SVC
Test Accuracy of SVC =  0.9691
```

After the classifier is trained, we save it and all relevant variables to a pickle file, to be used by the detection program:

```
with open('svc.p', mode='wb') as f:
    pickle.dump({'svc': svc, 'scaler': X_scaler, 'orient': orient, 'pix_per_cell': pix_per_cell, 'hog_channel': hog_c
                 'cell_per_block': cell_per_block, 'spatial_size': spatial_size, 'hist_bins': hist_bins,
                 'color_space':color_space, 'training_size':image_size}, f)
```

Detection

To detect cars in a image, I use the technique of **Sliding window**

Sliding Window Search

1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

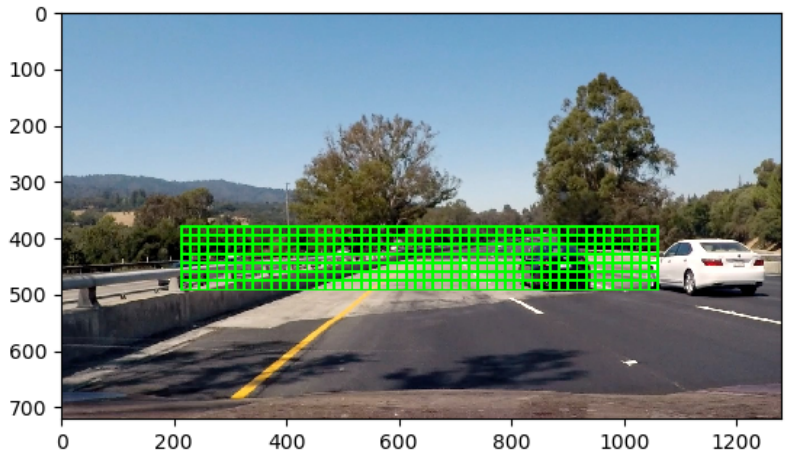
Our classifier is trained to recognize cars in small images, so we divide the image in *windows* and feed each one to the classifier to decide whether the window contains a car or not. This technique is known as *Sliding window search*. The first step is to construct a list of windows, defined by their coordinates. We need two points to define a window, two opposite corners. The code to construct a list of windows is in function `slide_window`:

```
def slide_window(img_shape=(1280,768), x_start_stop=[None, None], y_start_stop=[None, None],
                 xy_window=(64, 64), xy_overlap=(0.5, 0.5)):
    # If x and/or y start/stop positions not defined, set to image size
    if x_start_stop[0] is None:
        x_start_stop[0] = 0
    if x_start_stop[1] is None:
        x_start_stop[1] = img_shape[1]
    if y_start_stop[0] is None:
        y_start_stop[0] = 0
    if y_start_stop[1] is None:
        y_start_stop[1] = img_shape[0]
    # Compute the span of the region to be searched
    xspan = x_start_stop[1] - x_start_stop[0]
    yspan = y_start_stop[1] - y_start_stop[0]
    # Compute the number of pixels per step in x/y
    nx_pix_per_step = np.int(xy_window[0] * (1 - xy_overlap[0]))
    ny_pix_per_step = np.int(xy_window[1] * (1 - xy_overlap[1]))
    # Compute the number of windows in x/y
    nx_buffer = np.int(xy_window[0] * (xy_overlap[0]))
    ny_buffer = np.int(xy_window[1] * (xy_overlap[1]))
    nx_windows = np.int((xspan - nx_buffer) / nx_pix_per_step)
    ny_windows = np.int((yspan - ny_buffer) / ny_pix_per_step)
    # Initialize a list to append window positions to
    window_list = []
    for ys in range(ny_windows):
        for xs in range(nx_windows):
            # Calculate window position
            startx = xs * nx_pix_per_step + x_start_stop[0]
            # adjust last column to span all width
            if x_start_stop[1] - startx < xy_window[0]:
                endx = x_start_stop[1]
            else:
```

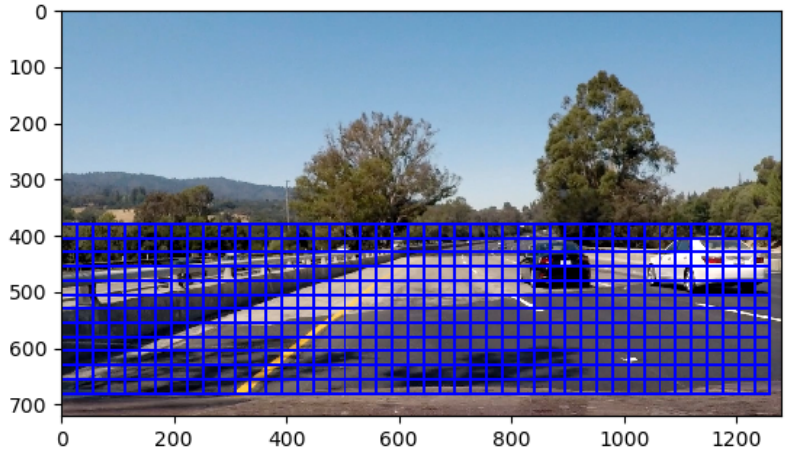
```
                endx = startx + xy_window[0]

            starty = ys * ny_pix_per_step + y_start_stop[0]
            # adjust last row to span all height
            if y_start_stop[1] - starty < xy_window[1]:
                endy = y_start_stop[1]
            else:
                endy = starty + xy_window[1]

            # Append window position to list
            window_list.append(((startx, starty), (endx, endy)))
    return window_list
```

The function takes the image dimensions (`img_shape`), the span to search in horizontal (`x_start_stop`) and vertical (`y_start_stop`), as well as the window size (`xy_window`) and the overlap from one window to the next (`xy_overlap`). The code starts by calculating the coordinates of the first window from `(xstart_stop[0], y_start_stop[0])` to `(x_start_stop[0]+xy_window[0], y_start_stop[0]+xy_window[1])` and then *slides* the window to the right so the next one overlaps `(xy_overlap[0]*100) %` of the previous one in horizontal. It continues like this until `x_start_stop[1]` , then goes down by `xy_overlap[1] %` and repeat the process.

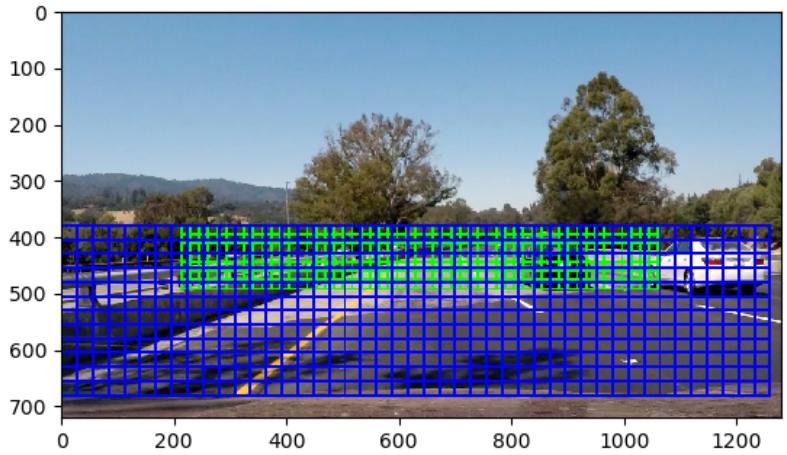
I used two sizes for the windows: small (64 by 64 pixels) in the upper part of the image for the cars that are far,



and a little bigger (112 by 100 pixels) for the lower part of the image where the nearer cars will appear.



The resulting grid looks like this:



Next step is to take each one of the windows, extract the features, and send them to the classifier to determine if there is a car or not inside. This is done in function `search_windows`:

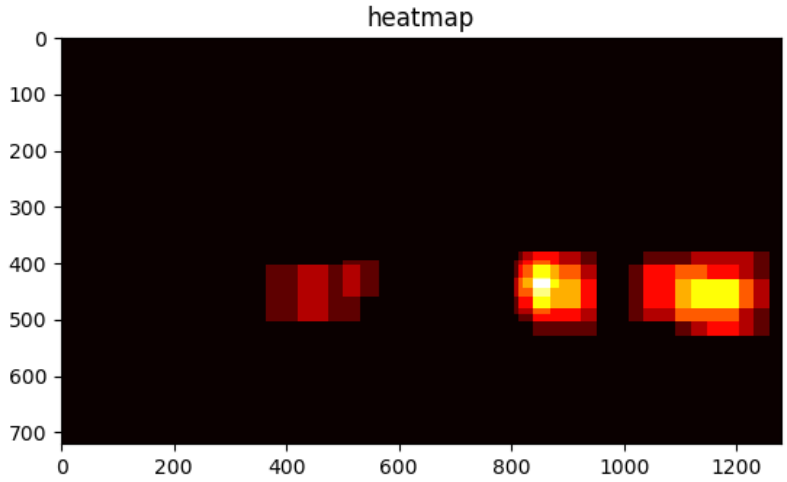
```
def search_windows(img, windows, clf, scaler, color_space='RGB',
                   spatial_size=(64, 64), hist_bins=32,
                   orient=9, pix_per_cell=8, cell_per_block=2,
                   hog_channel=0, window_size=(64,64), spatial_feat=True,
                   hist_feat=True, hog_feat=True):
    # 1) Create an empty list to receive positive detection windows
    on_windows = []
    img = convertColor(img, color_space)
    # 2) Iterate over all windows in the list
    for window in windows:
        # 3) Extract the test window from original image
        test_img = cv2.resize(img[window[0][1]:window[1][1], window[0][0]:window[1][0]], spatial_size)
```

```
        # 4) Extract features for that window using single_img_features()
        features = img_features(test_img, color_space=None,
                                spatial_size=spatial_size, hist_bins=hist_bins,
                                orient=orient, pix_per_cell=pix_per_cell,
                                cell_per_block=cell_per_block,
                                hog_channel=hog_channel, window_size=window_size,
                                spatial_feat=spatial_feat, hist_feat=hist_feat, hog_feat=hog_feat)
        # 5) Scale extracted features to be fed to classifier
        test_features = scaler.transform(features.reshape(1, -1))
        # 6) Predict
        prediction = clf.predict(test_features)
        # 7) If positive (prediction == 1) then save the window
        if prediction == 1:
            on_windows.append(window)
    # 8) Return windows for positive detections
    return on_windows
```

This function returns a list of the windows that the classifier deemed to contain a car. There may be more than one window over the same car, given that the search windows overlap. This will be used next to filter false positives, that is windows that were classified as containing a car but dont.

This filter process is implemented using a *heatmap*. A heatmap is a new image where the pixels inside the positive windows have a nonzero value, and everything else is 0. Each pixel contained in a positive window is increased by one; so if there are

pixels that correspond to two positive windows (overlapped) then those pixels will value 2. If there are three overlapped windows, the pixels will be 3, etc. Plotting the image assigning a different color to each nonzero pixel gives us a *heatmap* like the following:



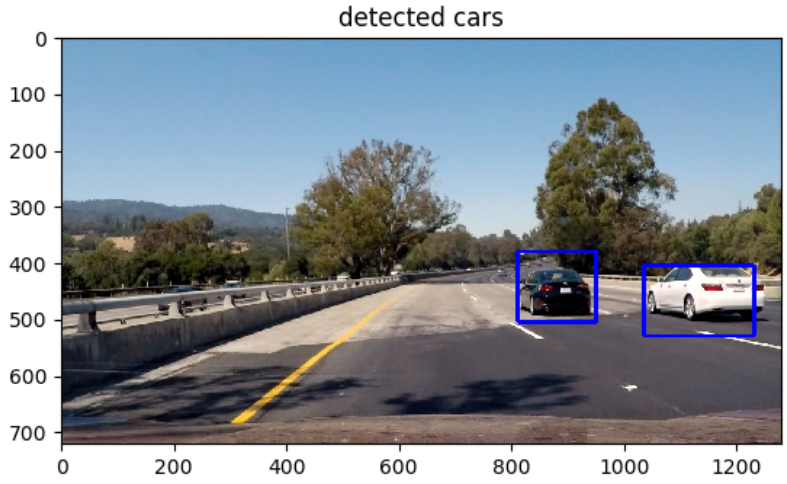
Usually, the classifier will get several windows over each car and single, isolated windows over different parts of the landscape, wrongly classified as cars (like in previous image). Once we have a heatmap of the image, we will have higher values inside overlapped windows (which almost surely contain a car), low values in isolated windows (misdetections), and zero everywhere else. We can apply a *threshold* to this heatmap to get only the higher-valued pixels:

<<< heatmap despues de eliminar los falsos positivos >>>

now we have regions of the image where the classifier 'thinks' there are cars. To group the overlapped windows we use the function `label` form scipy package that assigns a different label to each connected region, and draw boxes to enclose these connected regions in function `draw_labeled_bboxes()` :

```
def draw_labeled_bboxes(img, labels):
    # Iterate through all detected cars
    for car_number in range(1, labels[1] + 1):
        # Find pixels with each car_number label value
        nonzero = (labels[0] == car_number).nonzero()
        # Identify x and y values of those pixels
        nonzeroy = np.array(nonzero[0])
        nonzerox = np.array(nonzero[1])
        # Define a bounding box based on min/max x and y
        bbox = ((np.min(nonzerox), np.min(nonzeroy)), (np.max(nonzerox), np.max(nonzeroy)))
        # Draw the box on the image
        cv2.rectangle(img, bbox[0], bbox[1], (0, 0, 255), 6)
    # Return the image
    return img
```

The final image is



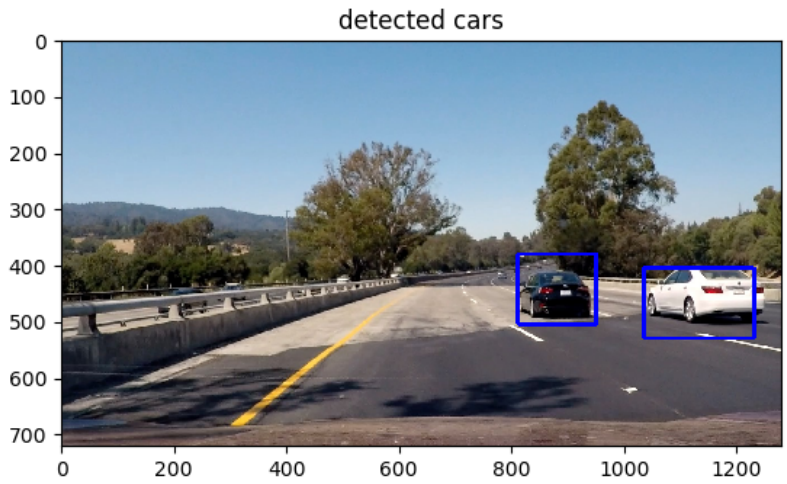
A special mention for videos: in a video, we can further filter the false positives by considering only those windows where there have been a positive detection on several consecutive frames. This is done in the code by storing the frames in a `deque` which is a first-in first-out data structure, in the code referenced by the variable `hm_history`. This is the complete processing pipeline for one image:

```
def process_image(image):
    global hm_history
    imgx = image.shape[1]
    imgy = image.shape[0]
    if len(windows)==0:
        # search boxes on two sizes, smaller for cars far away (up in the image), bigger ones for nearer cars
        windows.extend(slide_window((imgx, imgy), [imgx//6, 5*imgx//6], [380, 500], xy_window=(64, 64), xy_overlap=(0
        # window_img = draw_boxes(image, windows, color=(0, 0, 255), thick=4)
        # displayImage(window_img)

        windows.extend(slide_window((imgx, imgy), [None, None], [380, 700], xy_window=(112, 100), xy_overlap=(0.75,0.
        if debug:
            window_img = draw_boxes(image, windows, color=(0, 0, 255), thick=4)
            displayImage(window_img, title='slide windows')

    # classify content of search boxes using the trained linear svc
    hot_windows = search_windows(img=image, windows=windows, clf=svc, scaler=scaler, color_space=color_space,
                                 spatial_size=spatial_size, hist_bins=hist_bins, orient=orient,
                                 pix_per_cell=pix_per_cell, cell_per_block=cell_per_block, hog_channel=hog_channel,
```

```
                                spatial_feat=True, hist_feat=True, hog_feat=True)
    # intermediate result: bounding boxes on detected cars
    if debug:
        window_img = draw_boxes(image, hot_windows, color=(255, 0, 0), thick=4)
        displayImage(window_img, 'hot windows')

    # construct heat map to join multiple detection windows
    heat = np.zeros_like(image[:, :, 0]).astype(np.float)
    # Add heat to each box in box list
    heat = add_heat(heat, hot_windows)
    if debug:
        displayImage(heat, cmap='hot', title='heatmap')

    if single_image:
        heat = apply_threshold(heat, 3)
    else:
        hm_history.append(heat)
        if len(hm_history) == max_history:
            # Apply threshold to help remove false positives
            heat = apply_threshold(sum(hm_history), max_history+4)
        else:
            heat = apply_threshold(heat, 3)

    if debug:
        displayImage(heat, cmap='hot', title='hot with threshold')
    heatmap = np.clip(heat, 0, 255)

    # Find final boxes from heatmap using label function
    labels = label(heatmap)
    draw_img = draw_labeled_bboxes(image, labels)
    if debug:
        displayImage(draw_img, title='detected cars')
    return draw_img
```

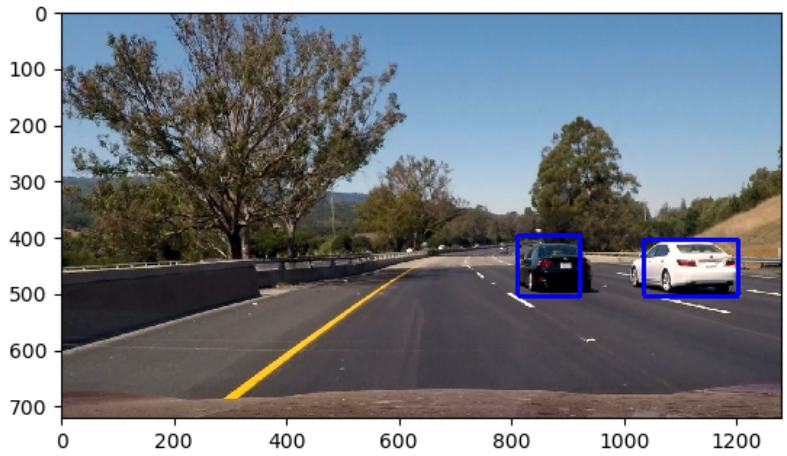
2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

Applying the pipeline of some test images render the following results:




0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200



Video Implementation

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (somewhat wobbly or unstable bounding boxes are ok as long as you are identifying the vehicles most of the time with minimal false positives.)

This is the result of applying the pipeline to the project video

2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.

It was described as part of the pipeline above.

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

I found this method difficult to apply in the real world. The detections are very dependent on the windows used in the search (size and position), and the cars will be of different sizes and be in different positions on each case. Also, the more windows we use, the more time it takes to perform the search. The classifier is not foolproof either: i had to do *a lot* of testing with different parameters and training sets to get a decent enough result. I think the detection would be better implemented using a convolutional network, an approach I would like to investigate. Another possible improvement is the features extracted from the images: there could be other features that can be more useful for this process, for example to Overall, this project involved a lot of trial and error and it still is not perfect as can be seen in the final video.